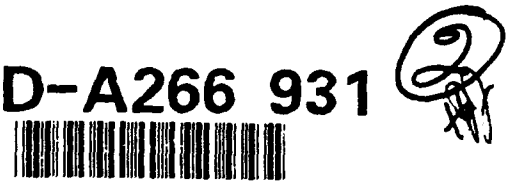
TO APPEAR: FTCS23



Relaxing Consistency in Recoverable Distributed Shared Memory

Bob Janssens and W. Kent Fuchs

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Abstract

Relaxed memory consistency models tolerate increased memory access latency in both hardware and software distributed shared memory systems. In recoverable systems, relaxing consistency has the added benefit of reducing the number of checkpoints needed to avoid rollback propagation. In this paper, we introduce new checkpointing algorithms that take advantage of relaxed consistency to reduce the performance overhead of checkpointing. We also introduce a scheme based on lazy relaxed consistency, that reduces both checkpointing overhead and the overhead of avoiding error propagation in systems with error latency. We use multiprocessor address traces to evaluate the relaxed consistency approach to checkpointing with distributed shared memory.

1 Introduction

Several parallel architectures use distributed shared memory to avoid the programming complexities of message passing. A distinguishing feature of these architectures is the distribution of memory across many processing nodes connected by an extensive network, resulting in high access latency for non-local data. The implementation of such a distributed shared memory can be in hardware and/or software. Hardware distributed shared memory systems have been developed in industry and academia [34]. All-software implementations that can be used on existing hardware, including networked workstations, have also been developed by researchers [8, 11, 22, 23]. To reduce the performance impact of remote memory access latency in distributed shared memory systems, relaxed memory consistency models have recently been developed [1, 7, 12, 15]. These models relax the traditional sequential consistency model to allow copies of data on separate processors to be temporarily inconsistent during the computation.

In this paper we show that, in shared-memory computer systems which require recoverability from transient node errors, relaxing consistency has the added benefit of decreasing the performance overhead of independent checkpointing and rollback recovery. We present checkpointing algorithms that take advantage of the conditions for relaxed consistency to reduce the minimum number of checkpoints required for correct operation. We also show how the checkpointing scheme reduces the cost of avoiding error propagation in systems with lazy relaxed consistency. We use multiprocessor address traces to evaluate the techniques by trace-driven simulation.

2 Background and Previous Work

2.1 Checkpointing in shared memory multiprocessors

In parallel systems, to ensure a correct global state, a rollback of one process often necessitates the rollback of other processes that have communicated with it. Schemes where processors globally coordinate checkpointing and rollback to limit this rollback propagation perform well where checkpoint intervals are relatively long [13], but become more expensive with shorter intervals. Other schemes that keep multiple checkpoints per processor to handle rollback propagation incur the additional overhead of dependency tracking. A more extensive discussion of the tradeoffs involved in the design of recoverable distributed systems can be found in the literature [13, 19, 31].

In shared-memory systems, processors communicate through shared variables rather than messages. Lee and Shin proposed hardware recovery blocks for recovery in a

This research was supported in part by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and by the Office of Naval Research under contract N00014-91-J-1283.

shared-memory multiprocessor [21]. In their approach, the detection of rollback propagation necessitates a complete restart of the whole task. Ahmed *et al.* used a flagged coordinated scheme to reduce the number of processors that need to participate in cache-based recovery [2]. Banâtre and Joubert proposed a recoverable multiprocessor that constructs global checkpoints with the help of a centralized stable memory device supporting atomic transactions [3]. Tam and Hsu have also worked on reliable distributed shared memory, concentrating on recovering the page table on a processor that has failed [29].

In distributed shared memory systems, synchronized global checkpointing is more costly than in traditional shared memory systems. To avoid coordination overhead, Wu and Fuchs used independent checkpointing at every communication between processors for cache-based recovery in bus-based multiprocessors [32] and for recoverable shared virtual memory [33]. Stumm and Zhou also used a similar communication-induced scheme in their fault tolerant distributed shared memory [27]. In the Sequoia multiprocessor [6], only data that have been checkpointed can be shared between processors. The schemes proposed in this paper use a similar communication-induced approach, but avoid checkpointing at every processor interaction.

Since checkpointing frequency with communication-induced schemes is typically high, they are suited for use with low-overhead state saving methods, based on the cache [2, 32] or virtual memory [9] mechanism. Under these methods, checkpointing usually consists of saving some registers and updating a counter. We assume that such a checkpoint can be committed immediately after a processor event, without the possibility of any intervening error. Unless assisted by a reconfiguration mechanism, our schemes only allow recovery from a single transient error in one of the processing nodes. The types of errors that can be handled depend on the exact state saving method used; the cache-based schemes are restricted to transient processor errors, while the virtual memory schemes can potentially recover from a crash of a complete processing node.

2.2 Sequential consistency and checkpointing

Previous work on recovery in shared-memory computer systems has assumed a sequential consistency model, which guarantees that all memory accesses appear to execute atomically and in program order [20, 24]. The model was developed to reflect the programmer's intuitive understanding of the correct execution of a multiprocessor. If sequential consistency is not maintained, synchronization and mutual exclusion using loads and stores to lock variables will not necessarily function correctly. Lamport defined sequential consistency as follows [20]:

> [A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

During normal execution, sequential consistency can be guaranteed by requiring processors to issue memory accesses in program order, and to wait until the result of a write access is made known to all processors before issuing any new accesses. Many commercial shared memory multiprocessors use processor consistency, a slightly more relaxed consistency model [16, 24]. The programmer's views under processor consistency and sequential consistency are similar, but processor consistency allows the hardware to pipeline memory accesses.

If checkpoints are taken at random times, a rollback on a recoverable system may violate sequential consistency. In the example of Figure 1, processor A sets variable x = 0, followed by x = 1 some time later. Then processor B reads x twice, storing the results in a and b. An error is detected by processor A in the time between the two reads on processor B, causing a rollback to the checkpoint taken before the second write. On processor B, the second read of x occurs before the rolled-back processor A reassigns x = 1. The outcome is that a = 1 and b = 0, which, if all operations were executed in sequential order, is only possible if the assignment x = 1 came before x = 0. This is not the order specified by the program on processor A, therefore sequential consistency is violated. If x were a lock variable, processor B would be able to acquire the lock, even though it was actually reserved by processor A.

To avoid the situation described, processor B could follow processor A in rolling back to a mutually consistent state. This strategy, however, would require coordination between processors during checkpointing. To avoid the overhead of coordination, a processor should not roll back past a write to a value that has been read by another processor. A processor could take a checkpoint immediately

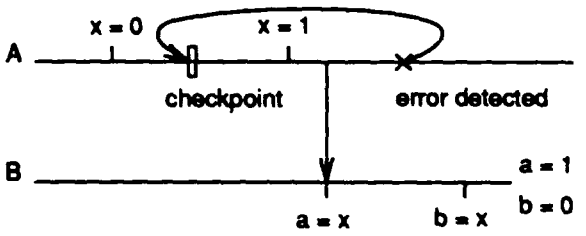


Figure 1: A violation of sequential consistency caused by a rollback past the read of a shared variable by a remote processor.

after every write, but then non-shared data would be unnecessarily checkpointed. A better strategy is to checkpoint the writer of a data item immediately after servicing a request from a remote processor to read that item [32, 33]. During recovery from an error, all reads from other processors must be delayed until the affected processor has rolled back to a correct state.

The overhead for this checkpointing scheme on a sequentially consistent system is high. A checkpoint is taken on every instance of communication between processors, leading to an unavoidably high checkpoint frequency [17]. If a read request from a remote processor occurs during rollback, the requesting processor needs to stall until the sending processor has completed recovery.

2.3 Relaxed consistency

Recently it has become clear that strong consistency requirements prevent systems designers from making certain performance optimizations. By using a more relaxed consistency model, accesses to remote memory locations can be delayed and reordered, increasing performance. One relaxed consistency model is weak consistency, where the following conditions hold [1, 12, 24]:

1. accesses to synchronization variables are sequentially consistent,
2. no access to a synchronization variable is issued in a processor before all previous data accesses have been performed,
3. no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

It is difficult to write and debug parallel programs that conform to the weak consistency model without additional constraints. However, Adve and Hill showed that a weakly consistent system appears sequentially consistent to a program as long as the program contains no data races and all synchronization accesses are visible to the memory system [1]. A data race occurs when two data operations on different processors access the same memory location, they are not both reads, and they are not ordered by a synchronization operation. It can be shown that for other relaxed consistency models, such as release consistency [15], similar conditions of visible synchronization accesses and data-race-free execution provide a sequentially consistent interface to the program.

Hardware that uses relaxed consistency has been developed by several researchers and implemented in the Stanford DASH multiprocessor prototype [15]. Many software distributed shared memory systems allow a user-controlled relaxing of sequential consistency [8, 11, 23]. Bormann and Herdieckerhoff first proposed *lazy* relaxed consistency to reduce the number of messages sent between processors in a software distributed shared memory system [7]. Keleher *et al.* showed with simulations that the performance of release consistency can be improved by implementing lazy release consistency [18].

No known previous work has taken advantage of general relaxed consistency to reduce the performance impact of checkpointing and rollback recovery. Some less general and more restrictive consistency models have been used in database and transaction processing applications. An extension to the Mirage shared virtual memory system, which uses a time-based coherence approach, has been proposed to make it recoverable [14]. The Sequoia multiprocessor [6], designed for transaction processing, uses a model similar to weak consistency to decrease checkpointing frequency. The current state is kept in the cache, which is flushed to main memory upon every acquire and release of a lock. The caches are not kept consistent between lock accesses. Although it can be accessed by user processes, Sequoia's shared memory is intended mainly for use by the operating system kernel.

3 Checkpointing with Relaxed Consistency

The conditions that allow a relaxed consistency system to appear sequentially consistent also allow a reduction in the number of checkpoints taken for recovery. If a program is data-race-free, a read of a data value by a remote processor is always separated from the write of that value by a write and subsequent read by the remote processor of a synchronization variable. Figure 2 presents an example, where a read of data item x by processor B is separated from the last write to that item on processor A by a write by processor A and read by processor B of synchronization variable s. A processor is guaranteed to never roll back past the write of a data item that is subsequently read by another processor if it can not roll back past the write to the intervening synchronization variable.

We present two schemes for low-overhead check-

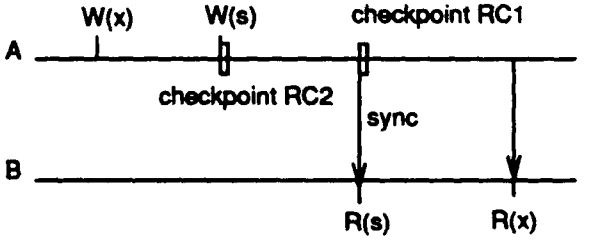


Figure 2: Checkpointing under the data-race-free model.

pointing with relaxed consistency, differing in the placement of checkpoints. Though the schemes are designed to work most efficiently with state-saving methods based on the virtual memory or cache (described in Section 2.1), they can be used with any other method. Rollback occurs completely independently on the affected processor; the other processors continue with the computation.

The first scheme, hereafter called RC1, uses a write-invalidate coherence protocol, where a write to a shared data item causes invalidation of all copies of that item on other processors. To guarantee correct rollback, the source processor is checkpointed only on reads by remote processors of synchronization variables. In the example of Figure 2, a checkpoint is taken after the read of s by processor B. This ensures that processor A will never roll back past the last write to data item x. Scheme RC1 reduces unnecessary checkpoints by checkpointing only writes to synchronization variables that are subsequently used to order accesses on other processors.

An alternative scheme is RC2. Here a checkpoint is taken immediately after every write to a synchronization variable, even if it is not read on another processor before the next write. Therefore the number of required checkpoints might be higher than in RC1. However, RC2 can be applied write-update protocols, which broadcast the new values of updated shared data. In Figure 2, scheme RC2 requires a checkpoint after the write to synchronization variable s on processor A, ensuring that processor A will never roll back past the last write to data item x.

Whether scheme RC1 or scheme RC2 is used, accesses to shared data variables never need to be checkpointed. Therefore the checkpointing overhead is reduced. Further decrease in checkpointing overhead can be accomplished by avoiding the use of shared memory for synchronization. Especially in software distributed shared memory systems, synchronization through shared memory is expensive. Many systems therefore implement synchronization primitives through message-passing [7, 11]. Contention for locks, and the resulting invalidations and data transfer, can then be avoided. For these systems we propose scheme RC3, which checkpoints immediately after the use of these synchronization primitives. A checkpoint is taken immediately after both the setting and unsetting of a lock. A checkpoint also needs to be taken by each participant immediately after passing a barrier used for synchronizing multiple processes.

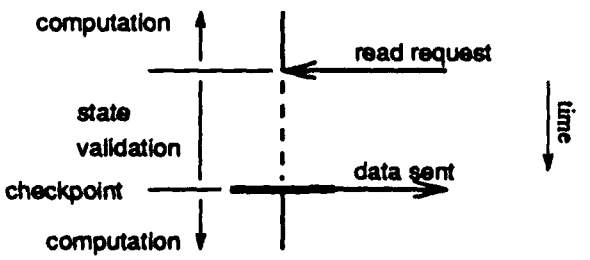


Figure 3: State validation and checkpointing during the handling of a remote read request.

4 Handling Error Latency with Lazy Relaxed Consistency

4.1 Error latency and error propagation

Work in the area of error recovery often assumes that errors are detected immediately and cause the affected processor to stop [2, 19, 26, 31, 33]. In practice, the overhead of error detection may be reduced by allowing some latency between an error and its detection. Even a few cycles of detection latency can dramatically improve performance, by taking error detection hardware out of the critical path [30]. If an error is still undetected when a checkpoint is taken, the checkpoint is corrupted. Wu *et al.* proposed a two-phase checkpointing scheme to solve this problem [32]. This solution, however, does not avoid the propagation of an error when an incorrect value is supplied to a remote processor before the error is detected.

To avoid propagating errors it is necessary to ensure the validity of data before supplying them to any other processor. Assuming the error latency is bounded, this validation could be accomplished by various methods. A local mechanism in the processor could ensure that the requested data item has not been updated for a time period equal to the error latency. For instance, the processor could cease computation and wait for a time period equal to the error latency before supplying the value. This method is very costly when error latency is high, and is therefore best suited for systems with low-latency error detection hardware [30].

Validation is needed both before the taking of every checkpoint, and before a response to a read request. In a sequentially consistent system, these events always occur together, leading to the order represented in Figure 3. Computation ceases as soon as the read request is received. Then validation occurs to prevent both the propagation of errors in servicing the request and the corruption of the following checkpoint. Then the remote read request is serviced, followed by the immediate commit of the checkpoint. The checkpointing process may have been initiated earlier, but the checkpoint has to commit atomically with the sending of the data before restarting computation.

With scheme RC1, checkpointing occurs only on synchronization reads from remote processors. However, vali-

dation still needs to occur as often as under sequential consistency: on every read from a remote processor. Therefore, the performance advantage of using relaxed consistency is reduced. With scheme RC2 and a write-invalidate policy, the validation overhead is even greater. In this case, communication and checkpointing occur at different times, and both need validation.

The performance overhead of state validation is exacerbated by the need for the processor that issued a remote read request to stall during the validation period, waiting for the requested data to be released by the sender. Furthermore, if an error detected during validation causes a rollback, the requesting processor may not continue until the rollback completes.

4.2 Lazy Relaxed Consistency

Relaxing consistency alone does not decrease the overhead of avoiding error propagation. A technique called lazy relaxed consistency, however, minimizes processor communication, thereby reducing the number of required computation state validations. With lazy relaxed consistency, propagation of newly written values is postponed until absolutely necessary [7]. As long as synchronization writes are used to order accesses, it is only necessary to update other processors with the results of all previous writes when a synchronization variable is written [1]. This delayed propagation strategy increases the performance of software distributed shared memory by reducing both the number of messages and the amount of data exchanged [18].

Figure 4 presents an example of the messages exchanged under lazy relaxed consistency. Each dashed arrow represents an update to a synchronization variable and a resulting write propagation. For simplicity only one variable, x, is written on processor A. The thin arrows indicate the value of x that would be returned by a read on the processor. Processor B only sees the updates to x after a write propagation from processor A.

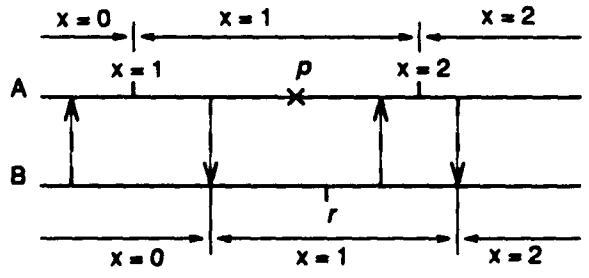


Figure 4: Values seen on processors using lazy relaxed consistency.

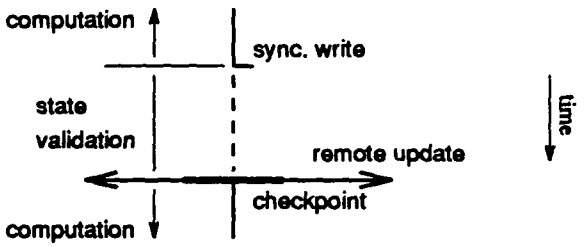


Figure 5: State validation and checkpointing during a synchronization write and remote update.

Lazy relaxed consistency is convenient for rollback recovery with error latency since error propagation can be constrained to write propagations. To achieve this, we describe scheme LRC2. A checkpoint is taken after every write to a synchronization variable, as in scheme RC2. A write-update coherence protocol is used where all the updates are performed in one message after a write to a synchronization variable. Errors can be propagated only during these updates. In the example of Figure 4 an error on processor A at point p can not propagate to processor B until the next write to a synchronization variable. Therefore the error at p will never propagate to a read at point r.

In scheme LRC2, a state validation and checkpoint occur immediately after a write to a synchronization variable. Figure 5 explains the sequence of events. After a synchronization write, computation stops and the computation state is validated. Then a remote update message is sent to all other processors. After the send, a checkpoint is taken to prevent the processor from a subsequent rollback past the remote update.

Lazy relaxed consistency protocols already send messages to update data at every synchronization points, so synchronization by message-passing can be easily implemented. Scheme LRC3, similar to RC3, can then be used. Checkpoints occur in the same instances as in scheme RC3. Every checkpoint is accompanied by a preceding state validation.

5 Simulation Results and Implementation Issues

The total overhead of the proposed schemes depends on the number of necessary checkpoints, the overhead per checkpoint, the number of required computation state validations, and the overhead per validation. Table 1 summarizes the various schemes, and gives expressions for their overhead, both with and without error latency.

We evaluate the alternative checkpointing schemes with

Table 1: Overhead of proposed schemes.

scheme	when necessary		overhead	
	checkpoints	validations	no error latency	error latency
SC	all reads by remote processors	all reads by remote processors	$rr * C$	$rr * (V + C)$
RC1	reads of sync. vars by remote procs.	all reads by remote processors	$srr * C$	$rr * (V + C)$
RC2	writes to sync. vars.	all reads by rem. procs. writes to sync. vars	$sw * C$	$rr * V + sw * (V + C)$
RC3	sync. events	all reads by rem. procs. sync. events	$se * C$	$rr * V + se * (V + C)$
LRC2	writes to sync. vars.	writes to sync. vars	$sw * C$	$sw * (V + C)$
LRC3	sync. events	sync. events	$se * C$	$se * (V + C)$

rr: number of reads by remote processors
srr: number of reads of sync. vars by remote procs.
sw: number of writes to sync. vars
se: number of synchronization events
C: overhead for one checkpoint
V: overhead for one validation

Table 2: Address trace characteristics.

program	description	tot. num. of references	data reads		data writes	
			total	shared	total	shared
gravsim[5]	N-body simulator	92,178,814	33,266,880	12,484,455	6,392,078	251,694
fsim[25]	fault simulator	149,918,375	50,950,933	39,326,911	3,958,919	999,127
tgen[25]	test generator	101,264,382	32,613,809	16,550,450	4,461,889	642,796
pace[4]	circuit extractor	87,861,165	23,266,576	1,286,787	7,842,338	348,524
phigure[10]	global router	132,998,231	38,244,233	4,281,207	11,530,981	1,876,400

results from simulations based on multiprocessor traces from five parallel scientific programs running on an Encore Multimax. The Multimax is a bus-based multiprocessor, supporting up to 20 processors. The traces are from execution on seven processors, and each contain at least 80 million memory references [17, 28]. Table 2 describes the characteristics of the traces used.

The programs available for tracing were specifically written for a tightly-coupled multiprocessor environment. In this environment, interprocessor communication and synchronization are not as costly as in a distributed shared memory environment. The programs therefore contain more synchronization overhead than they would if optimized for distributed shared memory. We therefore expect an actual implementation to exceed the performance improvements predicted by the simulations.

Figure 6 presents the frequency of various events of interest for the five programs. The overhead under sequential consistency amounts to one state validation and one checkpoint per read by a remote processor of a shared variable. Depending on the amount of sharing in the programs, reads by remote processors represent 400 to 4000 of every million accesses, or around 2600 on the average. Using the data-race-free programming model and scheme RC1, most checkpoints can be eliminated but state validations still occur on all reads by remote processors. Figure 6 shows that reads of synchronization variables by remote processors add up to only 50 to 1000 occurrences per million references, with an average of approximately 400.

Alternatively, RC2 or LRC2 can be used to take a checkpoint after every write to a synchronization variable. Synchronization writes occur more often than reads of a synchronization variable by a remote processor. For instance, when there is no contention for a lock, it is written twice (once to lock, and once to unlock) before possibly being acquired by another processor. Our results show an av-

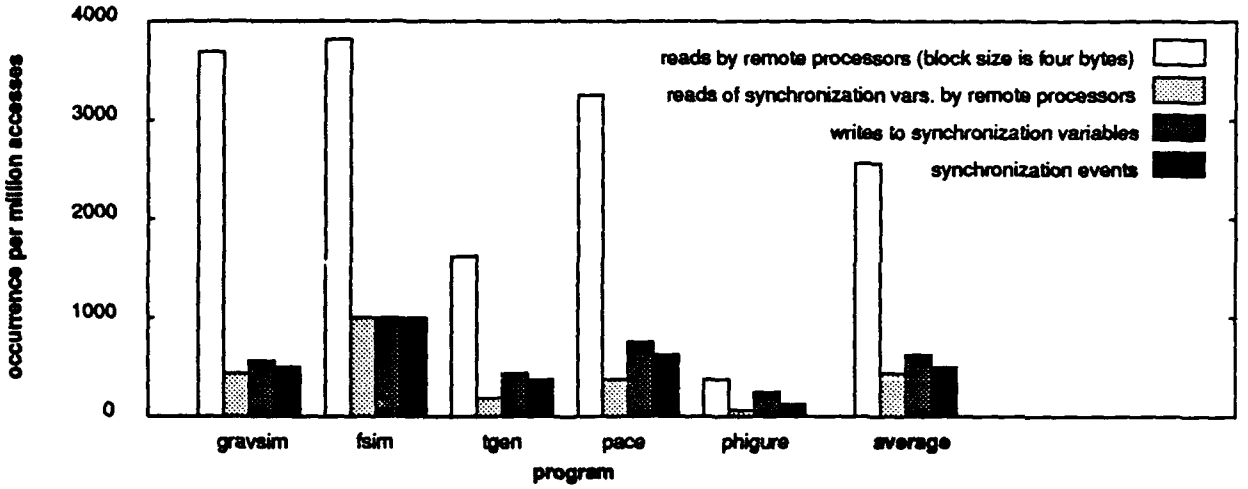


Figure 6: Events significant to checkpointing overhead for the five individual programs.

erage frequency of writes to synchronization variables of about 600 per million references. Implementing synchronization with message passing (RC3 and LRC3) further reduces the number of checkpoints required. Any writes to synchronization variables during contention for a lock are eliminated, and one checkpoint suffices per synchronization event. Figure 6 shows an average frequency of approximately 500 synchronization events per million references.

From the data in Figure 6 it appears more desirable to checkpoint on every read of a synchronization variable by a remote processor rather than on every write to a synchronization variable. However, the frequency of reads of variables by remote processors depends on the implementation of the distributed shared memory system. Interprocessor communication usually occurs in blocks or pages. It is necessary to checkpoint the source after a read by a remote processor of every block containing a synchronization variable. A larger block size causes different variables to be transferred between processors as a unit. Spatial locality, the tendency to reference variables in the same block as recently referenced variables, might decrease reads by remote processors. However, false sharing, where different variables in the same block are needed by different processors, might increase reads by remote processors. Synchronization variables are not usually accessed as a group, and therefore have very low spatial locality. False sharing, with other synchronization variables or with data variables, therefore increases the number of reads of synchronization variables by remote processors as the block size gets larger.

Distributed shared memory systems use a wide range of block size. A typical block size for hardware implementations is 16 bytes, while a typical block size for shared virtual memory systems is 4 kilobytes. In Figure 7, the average frequency of reads by remote processors is plotted against block size. As expected, false sharing and spatial locality interact, creating an optimal block size of 64 bytes to minimize reads by remote processors. Reads of blocks with synchronization variables by remote processors, on the other hand, continuously increase from an average of around 400 per million for four-byte blocks to around 1700 per million for four-kilobyte blocks. But the number of writes to synchronization variables and the number of synchronization events do not depend on block size. For all block sizes, except the unrealistic size of four bytes, checkpointing after writes outperforms checkpointing after reads by remote processors.

6 Conclusions

In distributed shared memory systems, the overhead of checkpointing and rollback recovery is increased by the need to handle rollback propagation. Unsynchronized checkpointing eliminates coordination overhead, but is costly in the number of checkpoints that are needed to ensure that the ordering of memory accesses does not violate sequential consistency. Under relaxed consistency models, only accesses to synchronization variables need to be strongly ordered. In this paper, we proposed new techniques that take advantage of this relaxed model to reduce the checkpointing overhead. Results from trace-driven simulations show a five- to ten-fold decrease in checkpointing overhead over previous techniques that require sequential consistency.

Lazy relaxed consistency allows delaying interprocessor communication until absolutely necessary for correct exe-

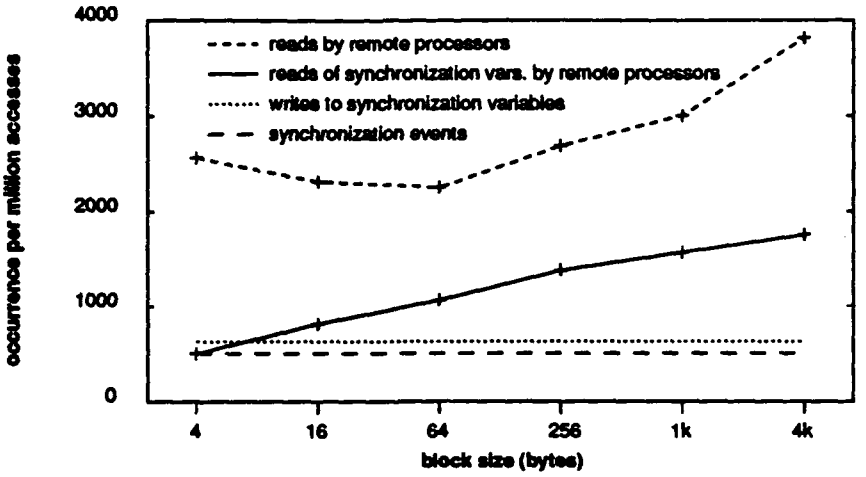


Figure 7: Events significant to checkpointing overhead versus block size.

cution. Updates to shared data can be bundled and transmitted at synchronization points. This decreases the number of messages exchanged and generates a communication pattern similar to that of explicit message passing. We took advantage of these properties of lazy relaxed consistency to reduce the overhead of avoiding error propagation in systems with unsynchronized checkpointing. We expect the communication properties of lazy relaxed consistency to be useful in implementing other recovery algorithms as well.

Acknowledgements

Lothar Borrmann and the Software Architecture for Parallel Computers group in the Systems Architecture Division at Siemens Corporate R&D, Munich, Germany, suggested many of the initial ideas on distributed shared memory used in this paper. At Illinois, Paul Chen, Shyh-Kwei Chen, Antoine Mourad, and Yi-Min Wang provided useful comments. Thanks also to Krishna Prasad Belkhale, Mark Bellon, Randy Brouwer, and Srinivas Patil for the programs that were traced to generate the simulation results.

References

[1] S. V. Adve and M. D. Hill, "Weak ordering—a new definition," *Proc. 17th Int. Symp. on Computer Architecture*, 1990, pp. 2–14.

[2] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems," *Proc. 20th Int. Symp. on Fault-Tolerant Computing*, 1990, pp. 82–88.

[3] M. Banâtre and P. Joubert, "Cache management in a tightly coupled fault tolerant multiprocessor," *Proc. 20th Int. Symp. on Fault-Tolerant Computing*, 1990, pp. 89–96.

[4] K. P. Belkhale, *Parallel Algorithms for Computer-Aided Design with Applications to Circuit Extraction*, Ph. D. Thesis, Tech. Report CRHC-90-15, Univ. of Illinois, Urbana, IL, Nov. 1990.

[5] M. Bellon, "Parallelizing gravitational N-body algorithms on MIMD architectures," Motorola Urbana Design Center, Urbana, IL.

[6] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 37–45.

[7] L. Borrmann and M. Herdieckerhoff, "A coherency model for virtually shared memory," *Proc. Int. Conf. on Parallel Processing*, 1990, pp. II-252–II-257.

[8] L. Borrmann and P. Istavrinos, "Store coherency in a parallel distributed-memory machine," *Proc. 2nd European Distributed Memory Computing Conf.*, 1991 (published by Springer-Verlag as Lecture Notes in Computer Science No. 487), pp. 32–41.

[9] N. S. Bowen, D. J. Pradhan, "Virtual checkpoints: architecture and performance," *IEEE Trans. on Computers*, Vol. 41, No. 5, May 1992, pp. 516–525.

[10] R. J. Brouwer and P. Banerjee, "PHIGURE: a parallel hierarchical global router," *Proc. 27th Design Automation Conf.*, 1990, pp. 650–653.

[11] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," *Proc. 13th ACM Symp. on Operating Systems Principles*, 1991, pp. 152–164.

[12] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," *Proc. 13th Int. Symp. on Computer Architecture*, 1986, pp. 434–442.

[13] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," *Proc. 11th Symp. on Reliable Distributed Systems*, 1992, pp. 39–47.

[14] B. D. Fleisch, "Reliable distributed shared memory," *Proc. 2nd IEEE Workshop on Experimental Distributed Systems*, 1990, pp. 102–105.

[15] K. Gharachorloo et al., "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Int. Symp. on Computer Architecture*, 1990, pp. 15–26.

[16] J. R. Goodman, "Cache consistency and sequential consistency," Tech. Report 61, SCI working group, March 1989 (also available as U. of Wisconsin Dept. of CS Tech. Report 1006).

[17] B. Janssens and W. K. Fuchs, "Experimental evaluation of multiprocessor cache-based error recovery," *Proc. Int. Conf. on Parallel Processing*, 1991, pp. I-505–I-508.

[18] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," *Proc. 19th Int. Symp. on Computer Architecture*, 1992, pp. 13–21.

[19] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 1, Jan. 1987, pp. 23–31.

[20] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers*, Vol C-28, No. 9, Sep. 1979, pp. 690–691.

[21] Y.-H. Lee and K. G. Shin, "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. on Computers*, Vol. C-33, No. 2, Feb. 1984, pp. 113–124.

[22] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, Vol. 7, No. 4, Nov. 1989, pp. 321–359.

[23] R. G. Minnich and D. J. Farber, "Reducing host load, network load, and latency in a distributed shared memory," *Proc. 10th Int. Conf. on Distributed Systems*, 1990, pp. 468–475.

[24] D. Mosberger, "Memory consisitency models," ACM Operating Systems Review, Vol. 27, No. 1, Jan. 1993, pp. 18–26.

[25] S. Patil, *Parallel Algorithms for Test Generation and Fault Simulation*, Ph. D. Thesis, Tech. Report CRHC-90-12, Univ. of Illinois, Urbana, IL, Sep. 1990.

[26] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Trans. on Computer Systems*, Vol. 1, No. 3, Aug. 1983, pp. 222–238.

[27] M. Stumm and S. Zhou, "Fault tolerant distributed shared memory," *Proc. 2nd IEEE Symp. on Parallel and Distributed Processing*, 1990, pp. 719–724.

[28] C. B. Stunkel, B. Janssens, and W. K. Fuchs, "Address tracing of parallel systems via TRAPEDS," *Microprocessors and Microsystems*, Vol. 16, No. 5, 1992, pp. 249–261.

[29] V.-O Tam and M. Hsu, "Fast recovery in distributed shared virtual memory systems," *Proc. 10th Int. Conf. on Distributed Computer System*, 1990, pp. 38–45.

[30] Y. Tamir and M. Tremblay, "High-performance fault-tolerant VLSI systems using micro rollback," *IEEE Trans. on Computers*, Vol. 39, No. 4, Apr. 1990, pp. 548–554.

[31] Y.-M. Wang and W. K. Fuchs "Optimistic message logging for independent checkpointing in a message-passing system," *Proc. 11th Symp. on Reliable Distributed Systems*, 1992, pp. 147–154.

[32] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 231–240.

[33] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. on Computers*, Vol. 39, No. 4, Apr. 1990, pp. 460–469.

[34] G. Zorpette, "The power of parallelism," *IEEE Spectrum*, Vol. 29, No. 9, Sep. 1992, pp. 28–33.